Obfuscated BFT

Technical Report
IRIT/RT–2012-8–FR
May-2012

Ali Shoker*, Maysam Yabandeh†, Rachid Guerraoui†, and Jean-Paul Bahsoun*
*University of Toulouse III, IRIT Lab.
Toulouse, France
firstName.lastName@irit.fr
†EPFL, LPD Lab.
Lausanne, Switzerland
firstName.lastName@epfl.ch

***Abstract*—A major assumption underlies the correctness of Byzantine fault tolerant (BFT) protocols: failure independence. Yet, state-of-the-art protocols typically rely on inter-replica communication in order to preserve consistency, and they even implicitly require replicas to store *access information* about each other. This jeopardizes the assumption of independent failures, since intruders can sneak from one replica to another and compromises the entire service.**

In this paper, we explore the idea of *obfuscation* in the BFT context. We present a BFT protocol called OBFT, where the replicas remain unaware of each other. OBFT assumes honest, but possibly crash-prone clients. Beyond obfuscation, the protocol has interesting scalability features for it imposes equivalent load on replicas and distributes multi-cast and cryptographic tasks among clients. We evaluate OBFT on an *Emulab* cluster with a wide area topology and convey its scalability with respect to state of the art BFT protocols.

***Index Terms*—Distributed Systems, Fault Tolerance, Obfuscation, Byzantine Failures.**

I. Introduction

Byzantine fault tolerance [1] (simply BFT) is a replication technique with the aim of tolerating arbitrary failures. State-machine based services [2] are deployed on replicas in *partially synchronous* systems [3]. At most one third of the replicas are assumed to be faulty and that faults are independent [4], [5].

Classical BFT protocols rely on inter-replica communication to ensure one-copy semantics. This makes the protocols fragile since the replicas must share some access information about each others. Consequently, an attacker that breaks the security walls of one replica can equally compromise the other replicas. A *Distributed DoS* attack is a typical example of such threats.

The competition to improve the performance of BFT protocols, perhaps, made BFT researchers take for granted BFT assumptions, and they mostly discarded the enhancements regarding independence of faults. One exception is Q/U [6] that avoids interaction between replicas in the *normal regime*; however upon failure detection, the protocol uses inter-replica synchronization to recover.

In this paper, we propose a new BFT protocol, called *OBFT* (Obfuscated BFT), avoiding any direct inter-replica communication. No replica knows anything about the others. The client plays a crucial role in OBFT and we do assume clients can not be malicious; though they can fail by crashing. In fact, if replicas do not communicate, a malicious client can violate consistency. The client sends two different requests to two distinct subsets of the replicas and behaves against each subset as if there was a single request.

Despite the fact that clients are trusted, many challenges make the protocol non-trivial. First, clients can still crash. To ensure obfuscation, OBFT must tolerate a crashed client (without inter-replica communication) otherwise unique request ordering among replicas can be compromised by the other clients. Second, upon failure detection, recovery is needed. In OBFT, faulty replicas will be replaced by correct ones (this might change the *primary* replica also). Thus, correct replicas should preserve a unique configuration; and the clients should be kept updated with these information too. Third, OBFT handles contending clients that can force the copies of an object on different replicas to skew.

We argue that this assumption makes sense for applications where the customers are trusted members of the same organization; e.g., airline ticketing services that provide access to different agencies. In an airline ticketing system, the company hosts its service on a replicas that belong to distinct private/public cloud. It allows access for the ticketing agencies around the globe. The ticketing agencies access the airline service via their secured and trusted servers, which are viewed as trusted clients by the BFT airline service.

Being mainly designed for WANs, OBFT looks like a good candidate to be deployed on clouds of different vendors and locations. Figure 1 conveys the idea behind our approach over clouds: the clouds are oblivious to the replication procedure, driven by the trusted client. An attacker hosted in one of the clouds cannot obtain the address of other replicas by compromising the replica inside that cloud or by monitoring its traffic. In fact, an attacker will not even know about the replication factor used by the client. The failure of a replica, hence, remains independent from the failure of other replicas, since the attacker could not locate the other replicas after compromising one of them.

OBFT requires $3f+1$ replicas in order to tolerate f Byzantine faults. The client in OBFT communicates with $2f+1$ *Active* replicas in its *Speculative* phase. The client sends a request to a primary replica that executes it, assigns it a

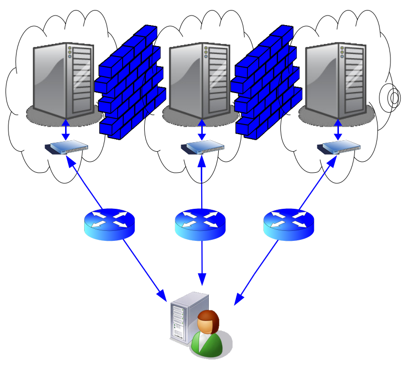
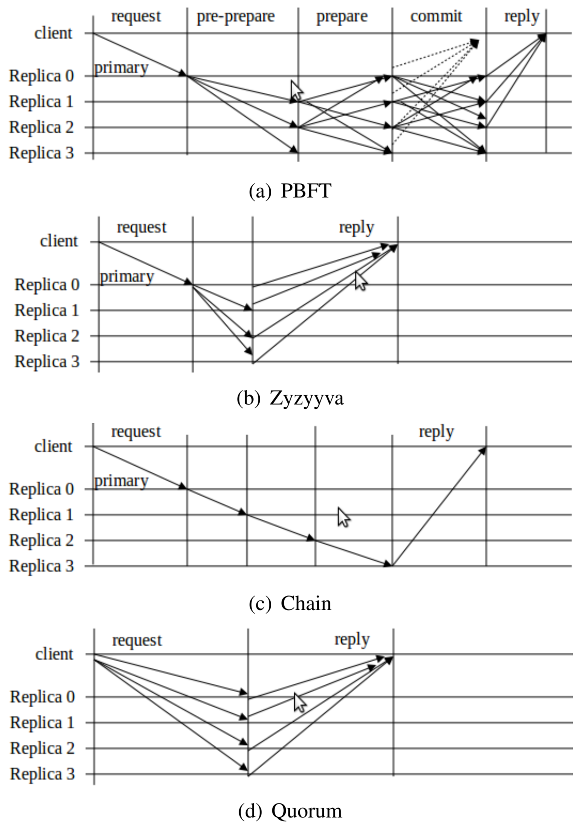


Fig. 1. Obfuscated BFT Scheme.



(a) PBFT

(b) Zyzyyva

(c) Chain

(d) Quorum

Fig. 2. BFT protocols message patterns for $f = 1$.

sequence number, and sends it back to the client. The client forwards the request to the other replicas that reply again to the client after executing the request command. The client accepts the operation if all responses match. Otherwise it launches a *Recovery* phase on the $3f+1$ replicas to exclude the *Suspicious* replicas (either faulty or slow); and then resumes to the Speculative phase acting on the new $2f+1$ Active replicas in the current view.

OBFT exhibits further interesting characteristics; namely, it achieves a good scalability as compared to state of the art protocols: (1) It reduces the load on replicas by pushing multi-cast and expensive cryptographic operations off the clients, and (2) it imposes almost identical load on the replicas.

We experimented OBFT on Emulab [7]. We setup 64-bit Xeon machines with 2 GB of memory in a star topology: each machine hosting a Debian OS. On such system, OBFT scales to hundreds of clients, and its peak throughput significantly exceeds that of state-of-the-art client-based protocols Q/U, and Quorum [1]. Also, we conduct some experiments to compare OBFT with other primary-based protocols to convey the obfuscation cost by using OBFT. TODO: Obfuscation overhead.

The rest of the paper is organized as follows. The background is recalled in Section II. Section III presents OBFT. After presenting the evaluation results in Section IV, we discuss some related work in Section V, and then we conclude the paper in Section VI.

II. Background

We recall here some basic BFT concepts and protocols. Also we show that to ensure consistency, etiher replicas communicate directly with each other or the client needs to be trusted.

A. State-of-the-Art BFT Protocols

BFT protocols are replication-based solutions to the problem of tolerating arbitrary failures of software and hardware components. A BFT protocol can ensure safety and progress up to a subset of one third of faulty replicas. Some BFT protocols, like Q/U [6], require more replicas.

[1]We do not count other BFT protocols here since they are not client-based, and hence this hampers obfuscation.

Most BFT protocols (PBFT [5], Zyzzyva [8], and Chain [9]) rely on a special *primary* replica to order the requests of the clients to ensure one-copy semantics. Other protocols, like Q/U [6] and Quorum [9], have no such primary replica, and avoid interactions among system replicas (unless for some recovery phases). The different message patterns of the protocols (for one faulty replica f) are depicted in Figure 2. The message pattern of Q/U is similar to that of Quorum, with $5f+1$ replicas instead of $3f+1$.

In **PBFT** [5], the client sends the request to the primary replica. The primary assigns a sequence number (an order) to the request and forwards it to other replicas (Figure 2;(a)). The replicas broadcast the received ordered request among each other to ensure that the primary is correct (i.e., it sent the same ordered request to all replicas). The client accepts the replies if they match. In **Zyzzyva** [8], the client sends the request only to the primary. However, as the other replicas receive the request as well as its sequence number from the primary, they immediately (i.e., speculatively) execute it and send the reply to the client (Figure 2;(b)). The client accepts the replies if they all match. Otherwise, either the primary or some of the replicas are faulty. In this case, the first correct client can detect that and demand changing the primary.

Chain [9] is a speculative protocol that is implemented on the the Abstract platform [10]. Chain uses a primary to avoid contention. All other replicas are ordered in a chain fashion, and each one forwards the request to its successor. The last replica in the chain, i.e., the tail, sends the reply back to the client (Figure 2;(c)). Although this technique increases the end-to-end delay, the throughput improves as the number of MAC operations by each replica is close to one, i.e. the theoretical lower bound. After detecting the failure, the whole protocol aborts and the *abort history* is used to initialize another instance of BFT protocol. Chain is designed for LAN settings where the latencies between replicas are small. In WANs, the protocol loses its good performance.

Quorum-based BFT protocols such as **Q/U** [6] do not use a primary and the clients directly communicate with the replicas. Q/U requires $5f + 1$ replicas to tolerate f Byzantine faults. Nevertheless, clients can only contact a *preferred quorum* (of size $4f$+1) for optimum performance. This could result in outdated histories in some replicas, which induce the cost of a *synchronization* phase to the protocol. In this phase, the outdated replica requests the up-to-date history from f+1 other replicas (to ensure that the history is not manipulated by some faulty replicas). Thus, although Q/U avoids direct inter-replica communication, it relies on replica interactions to *repair* from failures whenever the same copies on different replicas skew. Q/U has the advantage of optimal communication rounds in non-contention cases. However, it becomes very complex, as well as expensive, in the case of client request contention. As a result, the protocol is not scalable to large number of clients, where the probability of contention is very high.

Quorum [10] is another client-based BFT protocol built on top of the Abstract platform [10] also. Similar to Q/U, Quorum has the minimum latency among different protocols under no contention. The protocol has the same message pattern as Q/U in its trivial phase (Figure 2;(d)), however it requires only $3f+1$ replicas to tolerate f Byzantine replicas (whereas Q/U requires $5f + 1$). Although the Abstract platform has significantly simplified the design of Quorum, it suffers from the same contention problem as Q/U. In the case of contention or a Byzantine behavior, Quorum aborts and recovers to a *Backup Abstract* [10].

Therefore, based on what have preceded, and as depicted in Figure 2, we highlight some of the points that are interesting to our work. First, most of the protocols are built on inter-replica communication which can be dangerous in case one of the replicas got attacked by an adversary (since other replicas will be equally compromised). Second, most of the state of the art protocols rely on a primary replica to order clients requests, this is a simple way to ensure consistency, however, it might cause a bottleneck on the primary replica that performs a lot of cryptographic operations and multi-cast tasks. Third, the majority of the protocols are optimized to work on LANs, and they use multi-cast that is not yet supported in WANs, and hence using unicast instead, drops down the good performance of the protocol. In the paper, we show how can our approach resolve these issues.

B. *Independence of Failures*

As mentioned above, independence of failures is a major assumption for the correctness of the BFT protocols. However, little works addressed this issue.

Diversity of System Components. [11] discussed how to make intrusion fault systems by using BFT technology; and tried to maintain some diversity in system components to leverage the levels of independence. The authors defined two concepts:

- Axis of diversity: a component of a system that may be diversified (for example, the operating system).
- Degree of diversity: the number of choices available for a specific axis of diversity (for example: Linux, Windows,...).

Therefore, the report [11] proposes to use different hardware, platforms, operating systems, libraries... for different replicas. The authors also suggest using multiple physical facilities to avoid natural disasters and other localized physical threats.

Fault Independence on Clouds. Another study in [12], categorizes the different levels of failure independence on different setups in the cloud. The study shows that the protocols that are sensitive to the replica-client delay, and loss, do not perform well since the clients are typically connected via a WAN. For availability zones, where the latency between the replicas is higher, PBFT offers the best performance. However, Q/U performs the best when replicas are geographically distributed, because of the absence of communication between the replicas.

The authors conclude that: in order to achieve the highest failure independence in the oblivious clouds setting, inter-replica interaction should be avoided, and thus, the only available options are: Q/U and Quorum.

III. Design of OBFT

A. *Infrastructure*

OBFT is designed to be deployed on WANs; preferably, clouds of distinct providers, different platforms, and located in geographically distinct locations around the universe. Server farms like public/private clusters and *Grids* can also be considered. Different clouds are connected via the Internet, and they are not required to identify each other. System replicas can be chosen from distinct cloud vendors seeking high independence. Such an environment is crucial for OBFT to maintain the BFT assumptions for independence of failures.

B. *Model*

Our system model complies with the traditional BFT model (e.g., PBFT [5]). We assume a message-passing distributed system using a fully connected network among nodes: clients and servers. The network may (not infinitely) fail to deliver, corrupt, delay, or reorder messages. Faulty replicas may either behave arbitrarily, i.e., in a different way to their designed purposes, or they just crash (*benign* faults). A strong adversary coordinates faulty replicas to compromise the replicated service. However, we assume the adversary cannot break cryptographic techniques like: collision-resistant hashes, encryption, and signatures.

Clients might fail by crashing, but we assume they do not behave maliciously (they are typically part of the same organization that delegates its IT to the cloud). Liveness, however, is guaranteed only whenever the system is *eventually* synchronous; i.e., during intervals in which messages reach their correct destinations within some fixed worst case delay.

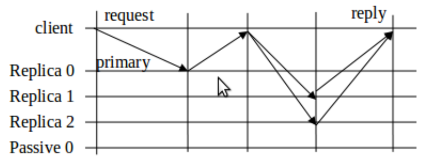


Fig. 3. Message diagram of OBFT running on three Active replicas and one Passive.

C. Algorithm

OBFT is a BFT protocol that avoids fault dependency among replicas and exhibits high performance in WANs. It avoids any inter-replica interaction; replicas communicate through the client instead. The protocol obtains a good performance using speculation: requests are executed speculatively on replicas. Cryptographic loads and requests multi-cast are pushed towards the clients to reduce the overload on replicas seeking better performance. OBFT uses *Abstract platform* [10] (also known by *Abortability*) techniques to recover upon failures.

OBFT, like most resilient BFT protocols, requires $3f$+1 replicas to tolerate Byzantine replicas, where no more than f replicas can be Byzantine. However, using $2f+1$ replicas only at a time, it can sustain faults, but cannot ensure progress. Thus, OBFT launches the *speculative* phase on $2f+1$ *Active* replicas. Upon failure detection, it recovers by replacing the *Suspicious* replicas (i.e., either faulty or slow) with correct replicas from the f *Passive* ones; and then, resumes to the speculative phase on a new Active set (in a new view). The $2f+1$ replicas are enough to collect a correct *abort history*. At any time, the protocol distributes the replicas over three sets:

- *Active* set: composed of $2f+1$ replicas (we call them *Active* replicas); these replicas are used in the *speculative* phase, speculatively.

- *Passive* set: composed of f idle replicas that are used as recovery backups.

- *Suspicious* set: This is a virtual set, composed of f replicas that are either faulty, or slow. In practice, it comprises the Passive replicas during the speculative phase, and possibly, a mixture of Active and Passive replicas during the recovery phase.

Particularly, upon failure detection, i.e., in *recovery* phase, the client identifies f Suspicious replicas, and replaces them with another f replicas (possibly from the Passive set or from the entire $3f+1$ replicas). Then, the new Active set becomes correct again in the new view, and the process continues as designed by launching the speculative phase.

Thus, OBFT algorithm consists of two main phases: a *speculative* phase, and a *recovery* phase. The messaging pattern of the speculative phase is depicted in Figure 3. In this section, we present the phases briefly (details are in later sections).

Speculative Phase. The communication pattern of OBFT in a failure-free scenario is simple, and it is concerned with the Active set only ($2f+1$ replicas):

1) The client first sends its request to the *primary*.
2) The primary assigns a sequence number to the request, executes it, and sends a reply back to the client along with the assigned sequence number.
3) The client then sends the request together with the assigned order (previously done by the primary) to the remaining $2f$ replicas in the Active set.
4) Each non-primary replica executes the received requests by order, and returns the replies to the client.
5) The client commits the request only if all the responses of the Active replicas match; otherwise, the recovery phase is launched.

Recovery Phase. This phase takes place using both: Passive and Active sets ($3f$+1 replicas).

1) Once the timer of the client expires waiting for $2f+1$ matching replies, the client panics by sending a *Panic* message to the Active replicas.
2) Replicas, upon receiving the Panic messages, stop executing new requests and send an *Abort* message back to client with their signed *local histories*.
3) The client constructs an *Abort history* collected from the $f+1$ matching replies (more details later), and sends an $INIT$ request to all replicas ($3f+1$) along with the abort history.
4) The replicas execute the $INIT$ request (they append it to their local histories), and reply to the client with ACK_{init}.
5) As the client receives $2f+1$ matching ACK_{init} replies, it considers their corresponding replicas as Active, and the remaining f replicas as Passive (those are *Suspicious* replicas). The updated Active set becomes correct again (in the next view), and the speculative phase is launched.

D. Algorithm Details

We describe here the algorithm. For simplicity, we assume no contention on replicas, i.e., a single client is accessing the service. Contention is addressed in later sections. (Because of space limitations, some details, pseudo-code, and proofs are given in the companion technical report [13]).

Client Role. To mitigate fault dependencies; OBFT clients enroll important tasks. First, the client issues the request towards the primary that assigns a unique sequence number. This is crucial to maintain consistency among different replicas. When the client receives the assigned reply from the primary, it validates its contents by verifying the MAC. The client takes the grip again to resend the signed request to the other $2f$ Active replicas, however this time, accompanied with the sequence number. At that instant the client starts a timer, waiting for the replies.

The final decision is also taken by the client. Upon receiving $2f+1$ replies from the replicas before the timer expires; the client verifies their MACs and makes sure the replies are matching. If so, the client considers the request complete. Otherwise, the client launches a *recovery* phase by collecting an abort history, cleaning the Active set from *Suspicious* replicas, and switching to a new Active set in another view.

The additional load on the clients in OBFT is tolerable. In fact, we are moving the load from one single machine (the primary) towards a plenty of machines (the clients), and thus,

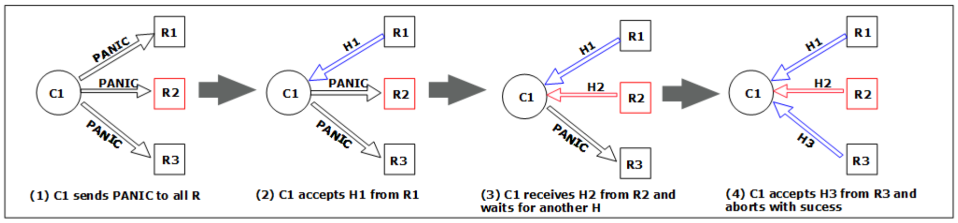

PANIC
PANIC
PANIC
C1
R1
R2
R3
H1
H2
H3
(1) C1 sends PANIC to all R
(2) C1 accepts H1 from R1
(3) C1 receives H2 from R2 and waits for another H
(4) C1 accepts H3 from R3 and aborts with sucess

Fig. 4. An example of OBFT while aborting, where $f = 1$.

all clients share slices of the task, that is usually assigned to the primary in other BFT protocols.

A Light Primary. OBFT pushes multi-cast and MAC overload towards its clients. In contrast to traditional BFT protocols, the primary in OBFT has almost the same load as other replicas. The only additional task is assigning an order to the requests which requires very simple computations. On the other hand, the primary is deprived from any multi-casting duties that can transform the primary to a bottleneck, especially that individual MACs should be computed to every replica. Instead, multi-cast is done by the client that orchestrates communication among replicas as mentioned before.

Ordering. Excluding the primary, all replicas validate the order of clients requests upon their receipt. They must verify requests with MACs and sequence numbers, and then execute them. The primary verifies MACs only. Replicas discard a request r_{new} in case $o(r_{last}) > o(r_{new})$; where r_{new} and r_{last} are the assigned orders of the current request and that of the last executed request in the local history of the replica, respectively. Each replica executes the request r_{new} if it has already executed all requests r_j where $o(r_j) < o(r_{new})$. Otherwise, request r_{new} is en-queued in a buffer, waiting for the missing requests that fill the gap. Final replies are authenticated via MACs and are sent by all replicas directly to the client.

Independent Replicas. To maintain independence of faults assumptions, obfuscation aspects requires the replicas to be unknown and unaware of each other. Replicas usually, need to communicate for two purposes: to ensure a total order (atomic) request execution, and to validate the response correctness (usually done by the primary). Thus, replicas in OBFT communicate with each other, however *blindly*, but guided by a mediator, the client. As depicted in previous sections, OBFT makes a unique total ordering possible through the client (that delivers the sequence number from the primary to the other replicas), though replicas cannot directly communicate. The other issue, i.e., validating a response, is also performed by the client upon checking whether all responses of the replicas are matching. Notice that assuming no malicious clients is essential for this fault independence to hold.

E. Recovery Phase

The *Recovery* phase is composed of three major steps: aborting, collecting abort history, and cleaning *Active* set from any *Suspicious* replicas.

Aborting. The client in OBFT considers a request as complete if the received $2f+1$ responses of the Active replicas are matching, before the expiry of the timer. Otherwise, the client stops sending requests and sends a *Panic* message to all Active replicas. Each replica, upon receiving the Panic message, stops receiving/executing requests, appends its *local history* of committed requests to an *Abort* message, and sends it to the client. The latter waits until it receives a sufficient number of signed Abort messages, i.e., the first $f+1$ non-conflicting ones. The intuition is that it is necessary and sufficient for the number of received correct commit histories to exceed the number of faulty ones (since no more f replicas can be Byzantine); knowing that faulty replicas might not respond at all. Aborting is achieved as follows:

1) The client waits for the first f+1 local commit histories to be received.
2) If no conflicting entries among the $f+1$ received local histories are found by the client, it stops receiving new histories, and collects the $f+1$ messages in a $Proof_{AH}$ set, that is used to form the abort history AH later.
3) Otherwise, if the client identified some commit histories with conflicting entries it waits for new local histories (since definitely there are correct clients that did not respond yet). The loop continues from step 1 again.

Figure 4 presents an example where the first two histories returned from replicas R_1 and R_2 are conflicting, consequently the client has to wait for the history from replica R_3. Thus, the phase continues till f+1 non-conflicting local commit histories are received by the client.

Building Abort History. A correct abort history is crucial for safety. It preserves total ordering and consistency across different switching phases (i.e., views). The abort history is collected from the current Active set, to initialize the local histories of the replicas on a new correct Active set. Building the abort history AH is done by the client after the receipt of $f+1$ non-conflicting signed abort messages from different replicas, collected in the $Proof_{AH}$ structure (as revealed before). The steps can be summarized as follows:

1) The client generates a history h such that: $h[j]$ equals the value that appears at position $j \geq 1$ of $f+1$ different local histories (LH_j), that appear in $Proof_{AH}$.
2) If such a value does not exist for some position x, then x is the last index of h.
3) Finally, AH is the longest prefix of h in which no request appears twice (i.e., exclude duplicate entries).

The resulting abort history AH thus includes all the globally committed client requests as well as some partially committed ones in the previous view; for example, if the request is received by at least f+1 replicas but not all of them (however this does not harm correctness). AH is used, then, to initialize the local histories of the new Active replicas (more details later).

Eliminating Faulty Replicas. The client in OBFT attempts to replace the faulty replicas in the Active set with correct ones from the Passive set; this is done by detecting the Suspicious replicas (faulty and slow ones) among the $3f+1$ replicas. This occurs as follows:

1) After collecting the abort history AH, the client sends an $INIT$ request (with AH appended) to the $3f+1$ replicas.
2) The replicas execute the INIT request, and append AH to their local history (if it is not already done through a previous view). Then, they reply to the client with ACK_{init}.
3) The client verifies the received ACK_{init} messages, and waits until it receives $2f+1$ matching ones.
4) The corresponding replicas to the first $2f+1$ matching messages constitute the new Active set, and the remaining f form the Passive set.

This process ensures eliminating the *Suspicious* replicas from the Active set, and having a correct one in the next view. Note that, all the exchanged messages among the replicas and the clients are authenticated/verified with MACs.

F. Dynamic Switching

Dynamic switching requires $3f+1$ physical replicas where only $2f+1$ of them are used at once as Active replicas. The other Passive replicas, however, are used to replace the Active replicas whenever failure occurs. This requires, sometimes, changing the primary replica also.

Initializing control information on clients. Upon launching the system, default Active set and primary replica are chosen. And since these are changing among the same physical replicas across different views, it is crucial for the client to know this *control information* before issuing its requests. The control information is collected on replicas and clients in the $CONTROL$ structure, that is composed of: (1) The Active set $ACTIVE$, (2) the primary P, and (3) the current view number $VIEW_{curr}$. Hence, some control messages are needed to deliver this information to any new coming client. The client gets informed as follows:

1) The client sends a GET_{info} message to all $(3f+1)$ replicas.
2) $Active$ replicas send an $INFO$ message to the client containing the control information: (1) The Active set $ACTIVE$, (2) the primary P, and (3) the current view number $VIEW_{curr}$.
3) The client waits until it receives $f+1$ matching $INFO$ messages from the replicas. Once done, it saves the information of the $INFO$ messages, updates its CONTROL, and starts sending its requests according to the protocol. Otherwise, it starts again from step 1.

Control information after recovery. However, when recovery occurs, and some client c succeeds in aborting, sending the $INIT$ request, and identifying the Suspicious replicas, the control structure CONTROL should change to exclude faulty replicas. Thus, the replicas refuse to execute any request until: a SET_{info} control message (see later) is received from c, or the timer expires. Instead, the replicas respond on any request type with an $INFO$ message (possibly with some empty fields).

In the case where the timer of a replica expires, it releases the lock, and accepts PANIC messages in the current view to allow other clients to perform a successful recovery [2]. SET_{info} is important after recovery by which replicas get assigned a new CONTROL. To fill the SET_{info} message, the client determines the Active set, selects the first replica to be the primary, and increments the view number.

If CONTROL on the replicas is already updated by the SET_{info} message, the INFO reply message will be complete and informative enough for the clients (upon receiving $f+1$ matching such messages). Otherwise, if the replicas are still waiting the SET_{info} message from c, the ACTIVE and the primary replica P fields in the INFO message will be empty, in contrast to the $VIEW_{curr}$ field that allows the clients to retry sending Panic messages in the current view successfully; since replicas do not accept any requests from other views. OBFT handles SET_{info} messages according to the usual communication pattern in Figure 3 (the receiving replica knows from the SET_{info} message itself, whether it is included in the Active set, it is a new primary, or it is a Passive replica).

G. Handling Contention

Views Perspective. Requests of any type are required to comprise the current view number $VIEW_{curr}$. In addition, any request received by the replicas is validated by verifying its MACs and the $VIEW_{curr}$ field, before being executed. Any request that belongs to a different view, gets rejected.

Panicking. The speculative phase in OBFT is deprived from contention problems as long as all requests that belong to the current view are ordered by the primary. However, upon failure, the clients launch the recovery phase. The recovery phase allows any client to panic, and to collect the abort history. Clients keep retransmitting $PANIC$ messages until they receive an enough number (i.e., $f+1$) of matching local histories, or a filled INFO messages from the replicas.

Initializing. When any client (one or more) creates the abort history, it sends $INIT$ message to all the replicas $(3f+1)$ to initialize their local histories and to update their view number $VIEW_{curr}$. Under contention, different replicas might receive different $INIT$ requests from different clients, and hence none will be completed since no client will be able to collect enough ACK_{init} from $2f+1$ replicas. Thus the clients follow an exponential back-off scheme that offers more chance that all replicas execute the same $INIT$ requests sent by some client c. This ends up by replicas, having consistent local histories in the new view.

Control Information. Afterwords, the replicas will be waiting for the SET_{info} request from the same client c. During this period, and to maintain non-conflicting messages from contending clients, any request will be discarded by the

[2]Recall that clients in OBFT can not be Byzantine, but they may crash.

replicas, that reply instead, by an $INFO$ message (where its Active set and primary replica fields are still empty) containing the $VIEW_{curr}$ that is needed by the clients while retrying their future request attempts. The step ends with a unique Active set and a primary replica across different views once the replicas receive the SET_{info} message from the client c. After this stage, any request from the clients will be handled as designed if it belongs to this view, otherwise, replicas respond with an $INFO$ message (where no fields are empty this time) to update those clients with the new control information. By this way, the clients can send requests as usual in the new view.

IV. Evaluation

In this section, we evaluate OBFT experimentally, and we provide some analytical evaluation as compared with state of the art protocols. Then, we focus on comparing OBFT with the obfuscated protocols Q/U and Quorum, being the main objective of the paper.

A. Experimental Setting

Our experiments are performed on 43 64-bit Xeon machines with 2 GB of memory employed on Emulab [7] cluster. No virtualization is used, thus simulating WAN environment on real machines. Each replica runs on a separate machine, and the clients are scattered over 40 machines (at least 3 client processes per machine). All machines are connected via a star topology. The maximum bandwidth of the network is set to 100Mb [3]. The end-to-end (E2E) delay is set to 20 ms and 60 ms depending on the setup (these represents a fast and an ordinary WAN speed, respectively).

For each setting, we have run four *a/b benchmark*[4] (same benchmark used in PBFT [5]) experiments using different payload sizes: 0/0, 0/1, 1/0, and 1/1. Without a payload, the size of the request and the reply messages are less than 100 bytes. The fault factor, f, is equal to one.

Multi-cast in PBFT, Zyzzyva, and Quorum is disabled (since they are deployed on WAN). Q/U experiments include a single difference where we needed six replicas (5f+1; for $f = 1$) instead of $3f + 1$, as this is the number required by Q/U [6] to operate.

B. Analytical Evaluation

Before proceeding with the experimental evaluation, we provide an analytical evaluation for OBFT as compared with the state of the art BFT protocols. The comparison is summarized in Table I. The results convey the interesting characteristics of OBFT as it achieves the lowest cost among BFT protocols in most cases (the bold entries in Table I). In the table, we consider most protocols optimizations (as in [8]), however, we exclude batching.

Row A in Table I shows that the number of replicas needed by OBFT to tolerate f Byzantine faults is minimal among

[3]The actual bandwidth that a client can use over a WAN is far below this limit.

[4]In a/b benchmarks, *a* and *b* correspond to request size, and response size in KB, respectively.

	PBFT	Zyzzyva	Q/U	Quorum	OBFT
A	**3f+1**	**3f+1**	5f+1	**3f+1**	**3f+1**
B	**2f+1**	**2f+1**	5f+1	3f+1	**2f+1**
C	2+8f	2+3f	2+4f	**2**	**2**
D	4	3	**2**	**2**	4
E	3	2	**1**	**1**	**1**

TABLE I
Analytic evaluation for the state of the art BFT protocols tolerating f faults, using MACs for authentication, and assuming preferred optimization (without batching): A represents the number of replicas needed to tolerate f Byzantine replicas; B is similar to A but excluding witness and backup replicas. C represents the number of MAC operations on the bottleneck replica. D is the number of one-way latencies needed for each request. E represents the number of send/to kernel calls on the bottleneck replica. Bold entries denote protocols with the lowest known cost.

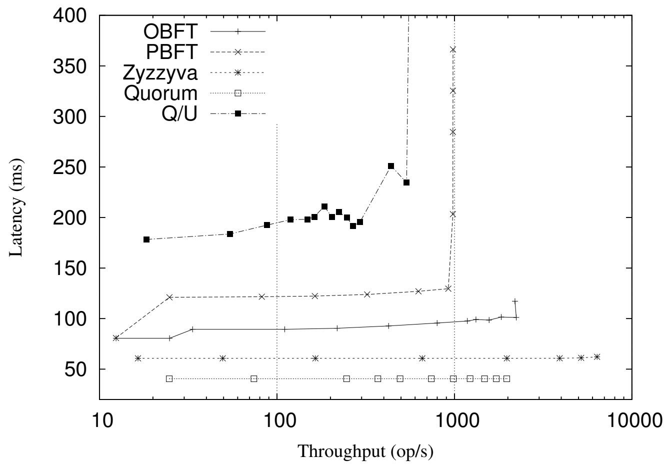


Fig. 5. Clients scalability of state of the art protocols for 0/0 benchmark in WAN setting, E2E=20ms.

the protocols. Similarly, the cost is minimal for OBFT in the speculative phase, since it communicates with only $2f + 1$ replicas out of $3f + 1$ (Row B in Table I). Moreover, Row C shows the number of MAC operations performed by the CPU of the bottleneck replica. The table points out that OBFT again has the minimal value 2, since any replica verifies the request only once, and authenticates the reply once.

Latency in OBFT (Row D of Table I) is not optimal as compared to other protocols; in fact, OBFT sacrifices this latency to maintain obfuscation. Finally, the number of $send$ (or $sendto$) calls to the system kernel is minimal on the bottleneck replica in OBFT. This is clear since every replica in the protocol sends the reply only once (Figure 3).

Thus, this analytical study shows that despite the latency cost that we *sell* to maintain independence of failures, OBFT characteristics appear to be equal or superior to that of the state of the art BFT protocols.

C. Obfuscation Cost

To leverage the fault tolerance of BFT protocols by mainlining obfuscation, it is worthy to pay some additional costs. The above analysis shows that the latency of OBFT is higher (Zyzzyva, Q/U, and Quorum) or equal (PBFT) to state of the art BFT protocols. Consequently, this impacts the throughput

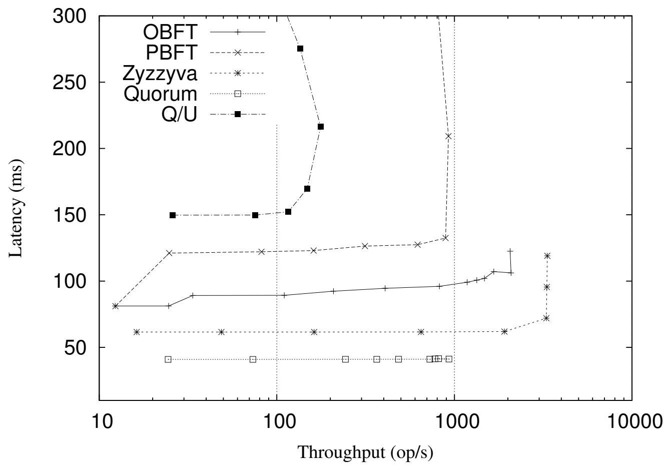


Fig. 6. Clients scalability of state of the art protocols for 1/0 benchmark in WAN setting, E2E=20ms.

of OBFT also. Despite this, OBFT maintains a good client scalability. Figures 5 and 6 convey the client scalability of OBFT with the state of the art protocols.

Figure 5 shows the scalability on the 0/0 benchmark, where end-to-end latency (E2E) is 20ms (we use logarithmic scale so that PBFT and Q/U can be observed). As the figure shows, Zyzzyva achieves the best scalability. It scales up to 400 clients reaching a peak throughput of 6365 op/s (for more than 400 clients, the protocol drops almost half of the packets), whereas, OBFT scales to 280 clients with a peak throughput equals to 2193 op/s. This is reasonable because of the simple three-delay message pattern of Zyzzyva (Figure 2). PBFT also, scales to 200 clients, however, its throughput is almost half that of OBFT (980 op/s); perhaps this refers to the extensive load of messaging on the replicas that PBFT imposes.

Q/U scales to 60 clients, scoring a throughput of 537 op/s. Beyond this point, the protocol suffers from the recovery phases due to contention. Note that, the latency of Q/U is lower than what is shown on Figures 5 and 6; we got these numbers, however, since we are running three client processes per machine, which greatly impact Q/U. Quorum scales to 80 clients only. We expected this, since Quorum is designed to work on LANs using multi-cast, which we disable in our WAN setting.

On the 1/0 benchmark as plotted in Figure 6, the scalability is quite similar for OBFT and Quorum (240 clients, and 60 clients, respectively), and no significant changes in throughput are noticed. However, Zyzzyva's scalability drops to 320 clients. Also, its throughput gets affected by the large payload (1KB) of the requests, and its peak performance degrades to 3332 op/s. We explain this to be caused by the bottleneck on the primary replica in Zyzzyva as requests get larger. OBFT is less affected by this since the operations required from the primary are negligible (multi-cast and cryptography).

Q/U and PBFT are the most affected protocols by increasing the size of the requests. As shown in Figure 6, Q/U scales to 30 clients only. The throughput, after that, starts to drop dramatically; again because of the recovery overhead under contention. However, PBFT scales to 120 clients in 1/0 benchmark. We refer this to the large number of messaging needed by the protocol (Figure 2).

D. Obfuscated Protocols Comparison

Since this paper is concerned mainly with obfuscation; we exclude PBFT and Zyzzyva in later sections, and we stick with Q/U and Quorum being client-based and can maintain obfuscation.

OBFT can be efficiently setup on WANs; preferably, on *cumulus clouds*, scattered on distant locations, and maybe from different vendors, to ensure higher levels of failure independence. In addition, OBFT acquires a good performance through various characteristics: (1) it needs only $3f$+1 replicas to tolerate f arbitrary faults (though speculative case communication is done on $2f+1$ Active replicas only at a time). (2) It relies on the clients to multi-cast request and not on replicas. (3) It pushes the cryptographic operations towards the client.

Assigning such jobs to the clients of OBFT does not impact their performance greatly, as we are moving the load away from the primary, i.e. a single machine, and distributing it on the clients (multiple machines), and thus, the clients will share small slices of additional work. OBFT design is important since, in any replication service, the replicas are, basically, the most critical part of the service.

Q/U [6], requires at least $5f$+1 replicas to tolerate f Byzantine faults. Additional fees shall be paid with larger f. Despite the use of preferred quorums (of size $4f$+1), Q/U provides less throughput than OBFT. This also makes Q/U more susceptible to failures under which its performance drops dramatically. This becomes lucid when the number of clients increases; partly because Q/U does not use a primary to order requests as OBFT does. In addition, although Q/U is client-based, it enforces inter-replica interaction upon failures, which makes replica failures more dependent. Note that, as mentioned above, Q/U can overcome this by not using preferred quorums, and hence, reducing its performance further. Regarding *fault scalability*, Q/U exhibited [6] great performance over agreement-based protocols; we expect Q/U to dominate OBFT in this sense, though we did not conduct experiments for $f > 1$.

Quorum [9] also shares some aspects with Q/U; mostly since it is client-based and involves only two communication phases. However, Quorum also suffers from interference under contention; this makes it hard to deploy on reliable contended services. In addition, Quorum is designed to multi-cast requests to the replicas. Of course, multi-cast in not yet supported on WANs, that is why we disable this feature (in fact, this causes some drop in performance). Quorum operates only in free-failure environment, and needs a recovery phase upon failure that might violate obfuscation.

1) ***Recovery Cost:*** Similar to Quorum, OBFT is inherently speculative and perform well only in best cases, i.e., when there are no faults. Under failures the protocol should abort to another Active set, and this imposes additional costs represented by *switching delays*. Switching delays are proportional to the end-to-end latency (E2E) of the WAN (assuming the execution time of the operation on the CPU is negligible

Protocol	Q/U	Quorum	OBFT
Message delays	2	2	4
Switching Message delays	-	4	4
Latency for E2E=20ms	41ms	40ms	80ms
Latency for E2E=60ms	121ms	120ms	240ms

TABLE II
Micro-benchmark latencies on WANs; where the end-to-end latency (E2E) is 20ms, and 60ms.

as compared to E2E). Table II (second row) conveys the number of message latencies needed to switch. The table shows that Quorum and OBFT need, in best case, four message delays to switch (i.e., the sum of $PANIC$, $ABORT$, $INIT$, and ACK_{init} latencies). The cost of switching can then be approximated by $4 * E2E$. We do not measure recovery cost for Q/U, since we assume using no preferred quorums (to ensure obfuscation).

*2) **Micro-benchmark**:* We present here the results on a benchmark, where only one client is accessing the replicated service. Table II displays the latency results (using 0/0 benchmark) for OBFT, Quorum, and Q/U by setting the E2E to 20 ms, and 60 ms.

When the E2E latency is set to 20 ms, OBFT achieves a latency of 80 ms; Q/U on the other hand reaches half this latency as depicted in Table II. We roughly relate this difference to the number of communication round-trips needed to complete an operation (as shown in the same table). In fact, OBFT needs a couple of round-trip messages; one message is sent to the primary to establish request ordering, and another is sent to communicate with other replicas. Q/U, however, achieves this latency since it completes the operation in a single round-trip instead of two.

Again, since Quorum (like Q/U) needs only two one-way communication phases to commit a request in a speculative way, they share same performance in a contention-free environment, the results are shown clearly in Table II.

Similar results are obtained upon changing the E2E latency to 60 ms. As shown on Table II, the latency of OBFT becomes 240 ms. This was expected because the large E2E latency becomes the main impacting factor in the service. The table also conveys the fact that Q/U again achieves half this latency. As mentioned above, this can be demonstrated by the number of round-trip messages in the protocols.

Analyzing the above numbers, we notice that the latency can be, roughly, obtained by the number of round-trips needed for one request multiplied by the E2E latency. This means that the system delay is the major factor overhead in the communication; the operation execution and MAC handling times are almost negligible as compared to the E2E latency. Notice that, although Q/U client needs to contact at least 5 replicas (i.e., the preferred quorum [6]), this does not impact the latency as one might expect, and hence keeps Q/U leading OBFT in such experiments.

*3) **Peak Throughput**:* To experiment the peak throughput, we used up to 300 concurrent clients on WAN setting with E2E=20ms. Since the results are close, we do not mention the results for E2E=60ms. The throughput achieved by OBFT is

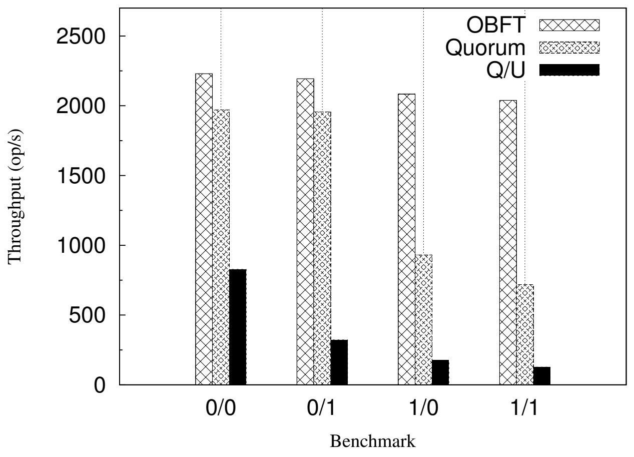


Fig. 7. Peak throughput for WAN setting with E2E=20ms.

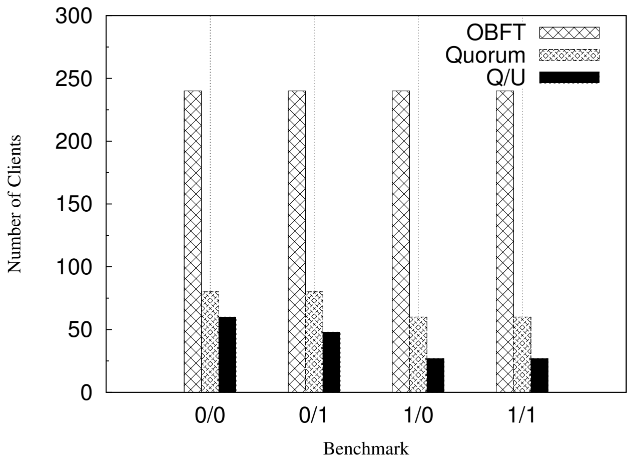


Fig. 8. Client scalability for WAN setting where E2E=20ms.

very interesting and inverts the leadership with Q/U which it exhibits in contention-free cases. As depicted in Figure 7, our protocol achieves a peak throughput of 2230 op/s for the 0/0 benchmark, the peak throughput of the other benchmarks (0/1, 1/0, and 1/1) are quite near as shown on the figure.

On the other hand, Q/U could not exceed 828 op/s throughput on the 0/0 benchmark (Figure 7). This peak throughput drops further as the request size gets larger. The benchmarks 1/0 and 1/1 of Q/U score no more 176 op/s and 127 op/s, respectively. These results can be justified since Q/U is not resilient to a high number of clients, and this forces the protocol to load excessive *Repair* and *Sync* phases, and the client *backoff* scheme. On the contrary to OBFT, that relies on the primary to order requests, and thus, avoids request collisions while accessing replicas, and pushes multi-cast and encryption overhead towards clients.

Quorum also gets affected by the request size more than OBFT. Figure 7 shows that the peak throughput of Quorum drops to a ratio of 1/2 whenever requests of 1KB are used (1/0 and 1/1 benchmarks). Despite this, in 0/0 benchmark, Quorum achieves a peak throughput close to that of OBFT; i.e., 1970 op/s. This refers to the simple message pattern of Quorum,

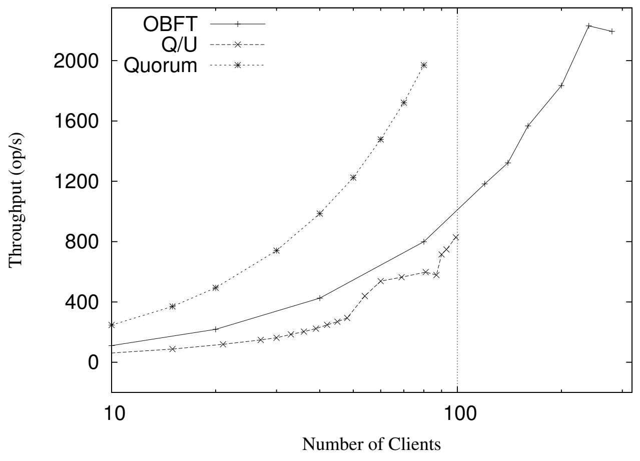


Fig. 9. Throughput of 0/0 benchmark, on WAN setting where E2E=20ms.

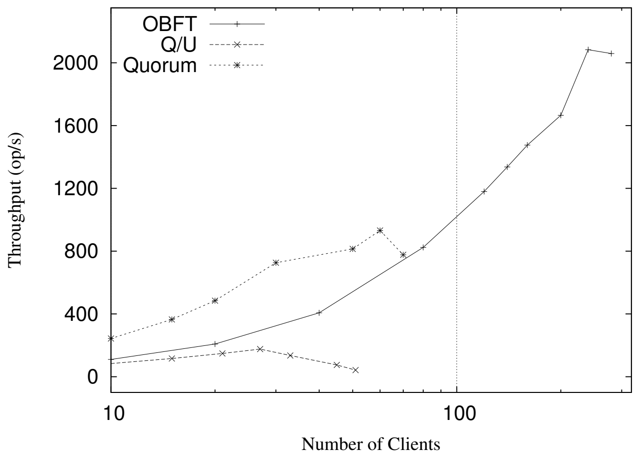


Fig. 10. Throughput of 1/0 benchmark, on WAN setting where E2E=20ms.

since it avoids the recovery phases needed by Q/U.

*4) **Scalability**:* Yet in another measure, i.e., client scalability, OBFT dominates Q/U and Quorum (Figure 8). In the experiments, the results of Q/U started to fluctuate for more than 30 clients. The protocol ceased to work for a number of clients greater than 30 or 80, depending on the experiment. Again, larger request sizes have a significant impact on the scalability of Q/U as the benchmarks 1/0 and 1/1 in Figure 8 show.

By observing Figure 8, we notice that Quorum also could not scale for more than 80 clients for the 0/0 benchmark, and 60 clients whenever larger requests are used. In fact, we do not expect more from Quorum, since it is designed to work on LAN and uses multi-cast that we disable in our WAN setting.

However, OBFT experiments finished successfully up to 240 simultaneous clients in all benchmarks (Figure 8). OBFT can handle this high number of clients since it avoids collisions among requests by having a primary replica to assign sequence numbers, and to distribute the load of multi-cast among the clients to avoid replica bottlenecks.

*5) **Throughput**:* Figure 9 plots the throughput results of OBFT for E2E=20ms (we use a logarithmic scale to better observe Q/U results). The x and y axes represent the number of clients and the throughput, respectively. The number of clients vary between 0 and 300 (starting with 3 client threads per machine). As depicted in the figure, the increase in the number of clients raises up the throughput of OBFT to 2230 op/s (for 240 clients). Then, as the number of clients increases, the throughput starts to degrade gradually.

As depicted in Figure 9, Quorum exhibits a better throughput than OBFT up to 80 clients. This is reasonable due to its simple message pattern. However, Quorum can not operate beyond that point because of the contention, since we use unicast instead of multi-cast in WANs. However, the throughput of Q/U remains below that of OBFT as shown in Figure 9, we refer this to high contention on Q/U as we use three client threads per machine.

Figure 10 conveys the results of 1/0 benchmark when setting the E2E latency to 20ms. The results are similar to 0/0 benchmark in Figure 9 for OBFT. However the throughput of Quorum and Q/U almost drops to a ratio of 1/2. We might explain this by the short message pattern of Q/U and Quorum (Figure 2) as compared to OBFT (Figure 3), where the delays in OBFT reduce the load on the replicas in WAN setting since the clients send requests through two phases; rather than Q/U and Quorum, that multi-cast the request directly to all replicas, in a single phase.

V. Related Work

Lamport introduced the problem of Byzantine generals in [1]. Then, he introduced the state machine replication approach using consensus in [2]. Liskov et al. [5] developed the first BFT protocol (PBFT) that can handle faults in asynchronous systems. The idea was similar to Paxos [14] (with benign faults) that uses views with a primary and backups to tolerate benign faults in an asynchronous system. However, PBFT had more complex message pattern and cryptographic authentication. In addition, PBFT required $3f+1$ replicas to tolerate f Byzantine faults ([4], [5]).

In the same work, i.e. in [5], the authors defined the model basics of BFT. In addition to asynchrony, they assumed independent node failures. They proposed that this can be achieved by using different operating systems, implementation (N-version programming), configuration, etc. Later works ([6], [15], [8], [9],...) then appeared to enhance the performance of PBFT. However, they all assumed the same proposition, i.e., independence of failures, and they did not work to enhance the protocols in such directions.

Few client-based protocols, such as Q/U [6] and Quorum [10] are then developed. These can maintain independence of failure via *obfuscation* (discarding the recovery phases need by Q/U). However, these notoriously do not scale with the number of clients. This is mainly because they are vulnerable to contention between multiple client requests, which makes the state of the replicas inconsistent. Upon detecting an inconsistency, the protocol has to call a recovery procedure to synchronize the replica states. Our tool OBFT has the same client-based design to ensure obfuscation among replicas; however, it benefits from a primary replica to handle contention, and hence, it is scalable to hundreds of clients.

Admitting that a single BFT protocol cannot fit all requirements, the notion of abortability [9] has recently been proposed to enable switching between BFT protocols whenever one could perform better, e.g., because of a change in the operating environment. Thus the authors develop an Abstract platform to make use of this abortability notion and to switch between protocols, when the load on the system changes. OBFT utilizes abortability by switching between Active sets: after a failure is detected, the protocol replaces the suspicious replicas by correct ones from the Passive set, and uses the updated Active set again in the next view.

Thus, few are the works that addressed independence of faults. [11] discussed how to make intrusion fault systems by using BFT technology. The study discussed how to maintain independence of faults by introducing diversity in system components, i.e., different hardware, software, operating systems,... The authors in [12] then categorized the different levels of failure independence on clouds, and proposed to deploy BFT services on *oblivious* clouds of different vendors and scattered in different geographical locations. In fact, OBFT is designed to work under such environments to leverage the levels of independence among replicas.

VI. Conclusion

This paper represents a step forward to leverage the reliability aspects of BFT replication. To respect the BFT assumptions of independence of failures, we introduce *obfuscation* among replicas so that they remain completely unaware of each other. We maintain obfuscation by using OBFT, a new client-based protocol deprived from any inter-replica communication, and designed mainly for WANs.

The design of OBFT allows to deploy BFT services on WANs, and especially clouds, to benefit from the versatility of the hardware, software, platforms... offered by different cloud vendors. In addition, OBFT can exploit the geographical distribution of the clouds around the globe to avoid natural disasters, and regional power failures...

Moreover, OBFT achieves a good performance as compared to state of the art client-based protocols (that can maintain obfuscation). Two simple design decisions are behind OBFT performance. First, we make use of a *primary* replica, avoiding contention between multiple client requests, to assign a sequence number to every request and thus ensuring one-copy semantics among all replicas. Second, we push the load of encrypting and multi-casting a request from the replicas, which are the bottleneck of agreement, to the issuing clients. Our experimental results show that OBFT scales to hundreds of clients in a WAN, while the throughput of state of the art BFT protocols quickly drops as the number of clients increases.

With few clients, the latency of OBFT, however, is higher than that of client-based protocols such as Q/U and Quorum [6], [9].

OBFT can be deployed on applications where customers are trusted members of the same organization; like airline systems with many agencies. This is needed since clients in OBFT are assumed not to be malicious, but they can crash. Thus, further work is needed to improve OBFT. We see that the most important point is to find a way to make OBFT tolerate malicious clients also.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, 1982.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] C. Dwork and N. Lynch, "Stockmeyer l: Consensus in the presence of partial synchrony," *Journal of The ACM*, 1988.

[4] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[6] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 59–74, 2005.

[7] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 255–270, 2002.

[8] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 45–58, 2007.

[9] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010, pp. 363–376.

[10] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocol," EPFL, Tech. Rep. LPD-REPORT-2008-008, 2008.

[11] Obelheiro Rafael R., Bessani Alysson Neves, Lung Lau Cheuk, Correia Miguel, "How practical are intrusion-tolerant distributed systems?" Department of Informatics, University of Lisbon, Tech. Rep. LPD-REPORT-2010-10, 2006. [Online]. Available: http://hdl.handle.net/10455/2992

[12] R. Guerraoui and M. Yabandeh, "Independent faults in the cloud," in *LADIS '10: Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*. New York, NY, USA: ACM, 2010, pp. 12–17.

[13] R. Guerraoui, M. Yabandeh, A. Shoker, and J. Bahsoun, "Obfuscating bft," EPFL, Tech. Rep. LPD-REPORT-2011-5, 2011.

[14] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[15] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: a hybrid quorum protocol for byzantine fault tolerance," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 177–190.

APPENDIX

In this Appendix, we explain OBFT further. We present a pseudo-code for the main functions of the protocol describing the two phases: *speculative* phase and the *recovery* phase; in both client and server sides. Also, we provide some correctness and progress proofs.

A. Notations

We denote the set of all $(3f+1)$ replicas by Σ. At any time $2f$+1 replicas are in the $Active$ set, and the remaining f are in the $Passive$ set. In addition, we denote the set of suspicious replicas (all faulty replicas, and possibly some correct ones) by $Suspicious$.

A message m sent by process p to the process q and authenticated with a MAC is denoted by $\langle m\rangle_{\mu_{p,q}}$. In addition, we denote the digest of the message m by $D(m)$. All processes are assumed to own the public key of every other process.

Notations for message fields and client/replica local variables used in OBFT are shown in Figure 11. To help distinguish clients requests for the same operation o, we assume that client c calls $Invoke(req, c)$, where $req = \langle o, t_c, c, V\rangle$, t_c is a unique, monotonically increasing clients time-stamp, and V is the view number. A replica r executes req by appending it to LH.

c	- client ID
P	- the primary replica
me	- the current replica
t_c	- local time-stamp at client c
$t_j[c]$	- the highest t_c seen by replica j
o	- operation invoked by the client
LH_i	- a local history at replica i
sn	- sequence number at replicas
AH	- abort history
V	- view number

Fig. 11. Message fields and process local variables

B. OBFT Pseudo-code

The pseudo-code of OBFT is presented in Figure 12. It includes the major functions of the client and the replicas. Some retransmission code and minor details are sometimes needed; however, we do not include this in the pseudo-code for clarity.

Fig. 12. Pseudo-code of OBFT

Pseudo 1 Join(c)

```
1:  {Sending to all}
2:  m ← ⟨ GET-INFO ⟩$_{\mu_{c,\Sigma}}$
3:  Send(m,i) $forall$ i ∈ Σ
4:  R ← $\phi$
5:  if $Timer_1() \neq \phi$ then {Until $Timer_1$ expires}
6:      while Receive(INFO,i) do {∀ i ∈ Σ}
7:          if vMAC(INFO) ∧ Match(INFO,R) then
8:              R ← R ∪{INFO}
9:          end if
10:         if $\|R\|= f$+1 then
11:             break
12:         end if
13:     end while
14:     {Update CONTROL info}
15:     CONTROL ← INFO.control
16: else {$Timer_1$ expiry}
17:     Join(c)
18: end if
```

Pseudo 2 Invoke(req,c)

```
1:  {Update Control Info.}
2:  Join(c)
3:  {Sending to Primary $P$}
4:  m ← $\langle REQ, req, V\rangle_{\mu_{c,P}}$
5:  Send(m,P)
6:  if $Timer_1() \neq \phi$ then {Until $Timer_1$ expires}
7:      Receive(REP,P) ∧ vMAC(REP)
8:      if REP.sn=NULL then {empty sequence #}
9:          Recover(req,c)
10:     end if
11: else {$Timer_1$ expiry}
12:     Recover(req,c)
13: end if
14: {Sending to the rest Active replicas}
15: for $i \in Active \setminus \{P\}$ do {Exclude P}
16:     m ← $\langle REQ, req, sn, V\rangle_{\mu_{c,i}}$
17:     $Send(m, i)$
18: end for
19: R ← $\phi$
20: R ← R ∪{REP}
21: if $Timer_2() \neq \phi$ then {Until $Timer_2$ expires}
22:     while Receive(REP,i) do {$\forall i \in Active \setminus \{P\}$}
23:         if vMAC(REP) ∧ Match(REP,R) then
24:             R ← R ∪{REP}
25:         end if
26:         if $\|R\|= 2f$+1 then
27:             break
28:         end if
29:     end while
30:     commit(req)
31: else {$Timer_2$ expiry}
32:     Recover(req,c)
33: end if
```

Pseudo 3 Init(AH,c,V)

```
1:  m ← $\langle INIT, AH, V\rangle_{\mu_{c,\Sigma}}$
2:  send(m,i) $forall$ i ∈ Σ
3:  R ← $\phi$
4:  if $Timer_1() \neq \phi$ then {Until $Timer_1$ expires}
5:      while Receive(ACK,i) do {∀ i ∈ Σ}
6:          if vMAC(ACK) ∧ Match(ACK,R) then
7:              R ← R ∪{ACK}
8:          end if
9:          if $\|R\|= 2f$+1 then
10:             break
11:         end if
12:     end while
13:     {Eliminate Suspicious replicas and choose a primary}
14:     CONTROL.P ← head(R)
15:     CONTROL.ACTIVE ← {R}
16:     CONTROL.PASSIVE ← Σ\{R}
17: else {$Timer_1$ expiry}
18:     return False
19: end if
```

Pseudo 4 Recover(req,c)

```
1:  Proof ← Panic(req,c)
2:  AH ← AbortHistory(Proof)
3:  CONTROL.VIEW ← CONTROL.VIEW+1
4:  init ← Init(AH,c,CONTROL.VIEW)
5:  if init!=TRUE then
6:      Recover(req,c)
7:  end if
8:  SetInfo(CONTROL,c)
9:  {Recovery success}
10: Invoke(req,c)
```

Pseudo 5 SetInfo(CONTROL,c)

```
1:  m ← ⟨ SET-INFO,CONTROL,V ⟩$_{\mu_{c,Active}}$
2:  send(m,i) $forall$ i ∈ $Active$
3:  R ← $\phi$
4:  if $Timer_1() \neq \phi$ then {Until $Timer_1$ expires}
5:      while Receive(INFO,i) do {∀ i ∈ Active}
6:          if vMAC(INFO) ∧ Match(INFO,R) ∧ full(INFO.control)
            then
7:              R ← R ∪{INFO}
8:          end if
9:          if $\|R\|= 2f$+1 then
10:             break
11:         end if
12:     end while
13:     {set CONTROL success}
14: else {$Timer_1$ expiry}
15:     Recover()
16: end if
```

Pseudo 6 AbortHistory(Proof)

```
1:  {Build abort history}
2:  tmp ← head(Proof)
3:  for all i: 1 to Proof.size do
4:      for all $LH \in Proof$ do
5:          if $\|tmp[i] = LH[i]\| \geq f + 1$ then
6:              h[i] ← tmp[i]
7:          else
8:              break
9:          end if
10:     end for
11: end for
12: {Remove duplicates}
13: for all i: 1 to h.size do
14:     for all j: i to h.size do
15:         if $h[i] \neq h[j]$ then
16:             $AH \leftarrow h[i]$
17:         else
18:             return $AH$
19:         end if
20:     end for
21: end for
22: return $AH$
```

Pseudo 7 Panic(req,c)

```
1:  {Send PANIC to Active replicas}
2:  m ← ⟨ PANIC,req,c,V ⟩$_{\mu_{c,Active}}$
3:  Send(m,i) $\forall i \in Active$
4:  R ← $\phi$
5:  if $Timer_1() \neq \phi$ then {Until $Timer_1$ expires}
6:      while Receive(ABORT,i) do {∀ i ∈ Active}
7:          if vMAC(ABORT) ∧ nonConflict(ABORT,R) then
8:              R ← R ∪{ABORT}
9:          end if
10:         if $\|R\|= f$+1 then
11:             break
12:         end if
13:     end while
14:     {Collected $f$+1 non conflicting histories}
15:     Proof ← R
16:     return Proof
17: else {$Timer_1$ expiry}
18:     return Panic(req,c)
19: end if
```

Pseudo 8 Server()

```
1:  while True do
2:      Receive(m,c)
3:      Handle(m,c)
4:  end while
```

Pseudo 9 Handle(m,c)

```
1:  {Handling client requests}
2:  if ¬ vMAC(m) $\vee m.t_c \leq t_i[c]$ then {request is not valid}
3:      return False
4:  end if
5:  if m.V ≠ CONTROL.VIEW then
6:      r ← ⟨ INFO,CONTROL ⟩$_{\mu_{me,c}}$
7:      send(r,c)
8:  end if
9:  if m.type=REQ then {Request message}
10:     if me is P then {if Primary}
11:         sn ← sn+1
12:         Exec(m) execute request, append to $LH$
13:         r ← ⟨ REP,rep,D(LH),sn,V ⟩$_{\mu_{me,c}}$
14:         Send(r,c)
15:     else {not a Primary}
16:         if m.sn= sn+1 then
17:             sn ← m.sn
18:             Exec(m) execute request, append to $LH$
19:             r ← ⟨ REP,rep,D(LH),sn,V ⟩$_{\mu_{me,c}}$
20:             Send(r,c)
21:         end if
22:     end if
23: else if m.type=PANIC then {Panic message}
24:     r ← ⟨ ABORT,D(LH),V ⟩$_{\mu_{me,c}}$
25:     Send(r,c)
26: else if m.type=INIT then {Init message}
27:     if LH = $\phi$ then {Empty local history}
28:         Exec(m) ∀ m ∈ AH
29:         LH ← AH
30:         C ← c
31:         lock ← True receive requests lock until timer expires or
            receive SET-INFO from C
32:     end if
33:     r ← ⟨ ACK,D(LH),V ⟩$_{\mu_{me,c}}$
34:     Send(r,c)
35: else if m.type=SET-INFO then {Update CONTROL}
36:     if c=C then {same client that sent INIT}
37:         CONTROL ← m.control
38:         lock ← False
39:         r ← ⟨ INFO,CONTROL,V ⟩$_{\mu_{me,c}}$
40:         Send(r,c)
41:     end if
42: else if m.type=GET-INFO then {Get CONTROL}
43:     {CONTROL can be filled or not}
44:     r ← ⟨ INFO,CONTROL,V ⟩$_{\mu_{me,c}}$
45:     Send(r,c)
46: end if
47: return True
```

C. Commit Certificate

Proposition. Any completed request by the client has been committed by the Active replicas.

Proof. The client in OBFT completes a request only if it has received $2f + 1$ matching responses including the local history digests $D(LH_i)$ of the Active replicas (Pseudo 2, lines: 23 to 26), among which $f + 1$ replicas are correct. Recall that, local histories (LH_i) are uniquely defined sequences of requests, which represent the replica state at any time. Then, since a correct replica appends the new request (upon execution) to its local history before sending the LH digest ($Exec$ method in Pseudo 9), then these digests represent an indication for the client certifying that its request has been committed successfully.

D. Validity

Proposition. Any request that is found in the commit/abort history must have been sent by some client.

Proof. A client commits a request only if all the received commit histories (LH) of the Active replicas are matching

(Pseudo 2, lines: 26 to 30). Thus, at least $f+1$ correct replicas must have executed the request, and appended it to LH. On the other hand, a replica executes (i.e., appends to LH) a request message REQ (or $INIT$ message) only after validating its sender identity; that should be some client. In addition, to avoid duplicates in LH, a replica always maintains and checks the last client time-stamp $t_i[c]$ (Pseudo 9, line 2).

As for the abort history AH, since it is collected from $f+1$ matching LH (Pseudo 6); thus, all requests in the AH are sent by some client in a previous view (follows from the previous paragraph). As for AH duplicates, they are removed by construction (Pseudo 6, lines: 13 to 21).

E. Termination

Proposition. Aborting from the $Specultive$ phase eventually occurs.

Proof. OBFT runs the Speculative phase until: (1) the client detects non matching responses from replicas or (2) its timer expires. In both cases, the client should abort the Speculative phase by sending a $PANIC$ messages to all the $Active$ replicas (for progress, it keeps sending such messages until receiving the needed $ABORT$ messages). The replicas should eventually receive the $PANIC$ messages (according to our assumption that sent messages are not infinitely delayed or dropped by the network). Thus, at least $f+1$ correct replicas should send $ABORT$ messages to the client (Pseudo 7, line 10). When the client (eventually) receives $f+1$ matching $ABORT$ messages, it aborts the request (since OBFT clients can not be malicious).

F. Lemma 1

Denote the state of the local history of replica r_i upon appending request req to LH_i by LH_i^{req}. Then, for any message m sent by r_i upon appending to LH_i with history LH_i^m, LH_i^{req} is a prefix of LH_i^m. In other words, LH_i^{req} remains a prefix of LH_i forever.

Proof. Let the current state LH_i of some replica r_i be LH_i^{req}. A correct replica r_i modifies its local history LH_i by sequentially appending any new request m to LH_i; in particular, appending to its prefix LH_i^{req} ($Exec$ function in Pseudo 9). Hence, $\forall$ new request m, LH_i^{req} remains a prefix of LH_i^m forever.

G. Commit Ordering

Proposition. Commit histories can not contain requests in conflicting orders.

Proof. Assume, by contradiction, that there are two committed requests req and $req' \neq req$ (sent by two clients c and c', respectively) with different commit histories h_{req} and $h_{req'}$, such that, neither is the prefix of the other. Since a correct client commits a request only when it receives $2f+1$ identical LH digests from replicas (Pseudo 2; line 26); then, there must be a correct replica r_j that sent $D(h_{req})$ to c and $D(h_{req'})$ to c' such that h_{req} is not a prefix of $h_{req'}$ nor vice versa. A contradiction with Lemma 1.

H. Abort Ordering

Proposition. For any committed request req, every commit history h_{req} is a prefix of any abort history AH.

Proof. Considering a single replica r_i; suppose that $\exists$ a request req and $ABORT$ message m such that h_{req} is not a prefix of LH_i^m. Since req is already committed, then it must be included in $2f+1$ local histories from the Active replicas including r_i (Pseudo 2; line 26). But r_i does not send $ABORT$ messages unless after it stops executing new requests in the current view; thus r_i executed req before m. Hence, by Lemma 1, h_{req} is a prefix of LH_i^m. On the other hand, since an abort history is constructed from $f+1$ matching LH digests sent by correct replicas (Pseudo 6), then h_{req} must be a prefix to all these LH, i.e., to the abort history AH.

I. Init Ordering

Proposition. INIT history is a prefix for any commit/abort history.

Proof. Every correct process must initialize its local history with some valid $Init$ history before sending any message (Pseudo 9, lines: 26 to 34). Since any common prefix CP of all valid $Init$ histories is a prefix of every single $Init$ history I, thus CP is a prefix for every local history sent by a correct replica. $Init$ ordering for commit histories immediately follows.

In the case of abort histories; f+1 correct $ABORT$ messages are received by a client upon aborting a request. The ABORT message contains the replicas LH (Pseudo 9, line 24) that have CP as a prefix. Thus, CP is a prefix of any abort history AH.

J. Getting Control Info.

Proposition. Any new client gets the correct CONTROL information from replicas.

Proof. New clients get CONTROL information from replicas to know the ACTIVE replicas, the primary P, and the current view number. This is done through sending a GET-INFO message to all replicas (Pseudo 1). The client accepts the CONTROL information only if it got f+1 matching INFO messages from replicas. This is enough since at most f replicas can be Byzantine. Thus new clients always receive correct CONTROL info.

K. Control Info. Uniqueness

Proposition. At any time, all correct replicas have a unique CONTROL Info.

Proof. Upon launching the service, a pre-defined CONTROL information is set by default on all replicas (where at least 2f+1 of them are correct). Then, during the Speculative phase, the replicas discard any SET-INFO message from clients, hence, preserving the uniqueness of the CONTROL information.

Upon failure detection, at least one client c panics and collects the abort history AH. The client c sends an INIT message to all replicas (Pseudo 3). Any replica that receives the INIT request executes it and refuses to accept any request, except a SET-INFO request from the same client c containing the new CONTROL information set by c [5] (Pseudo 9, lines: 26 to 41). The replicas reply back with an INFO message to the client that approves the SET-INFO upon receiving $2f$+1 matching INFO messages, or it panics again (Pseudo 5, lines: 5 to 15). Therefore, assuming clients can not be Byzantine, all correct replicas will have the unique CONTROL information sent by c.

L. Eliminating Byzantine Replicas

Proposition. Recovery phase ends with eliminating Byzantine replicas and replacing them with correct ones.

Proof. Upon failure detection, and panic, at least on client c sends INIT request to *all* replicas (Pseudo 7). Once the replicas receive INIT request from the same client c, they respond with an ACK message containing the digests of their local histories. The client only accepts ACK if it receives $2f$+1 matching such messages (Pseudo 3, lines: 9 to 16). Thus, at that same moment, the replicas that correspond to these ACK messages are correct (Active), and the remaining f replicas are considered Suspicious (comprises all Byzantine replicas and, probably, some correct ones).

M. Progress

Proposition. Clients eventually receive replies to their requests.

Proof. Recall: we assume that the network can not delay messages infinitely. Thus, we suppose any sent messages to reach its destination within a maximum delay δ'. Since the system is partially synchronous then we choose some unknown $\delta \geq \delta'$.

In the $Speculative$ phase of OBFT, clients wait for responses from replicas twice: (1) waiting for the primary ($Timer_1$, Pseudo 2, line 6), and then (2) waiting for the other $Active$ replicas ($Timer_2$, Pseudo 2, line 21). Adjusting $Timer_1$ and $Timer_2$ for a duration of 2δ+t (i.e, the delay for a complete request round trip + the expected execution time at replicas) ensures that the client will eventually receive $2f$+1 replies from the $Active$ replicas. Otherwise, the Recovery phase is invoked.

In the $Recovery$ phase, the client sends a PANIC message and waits for f+1 non-conflicting ABORT messages (Pseudo 7). Again by setting the timer of the client to 2δ+t, receiving the f+1 messages is ensured (the client keeps retransmitting PANIC messages until it receives the f+1 ABORT messages). This must occur since at least f+1 Active replicas must be correct. After that the client sends the INIT request along with the abort history AH and waits for $2f$+1 matching ACK messages. A timer in this case can be adjusted to $2\delta + t$ also to ensure progress. This represents the time for the INIT messages to reach the replicas + handling time at replicas + the delay of ACK messages from the replicas to the client (however this is not shown in the pseudo-code).

As the client c receives $2f$+1 matching ACK messages, it sends the SET-INFO message and starts a timer waiting for INFO messages from replicas (set to 2δ+t) as above. The replicas however, and after receiving the INIT requests, they decline to accept any request unless the SET-INIT request from the same client c. For that, they adjust a timer to 2δ+t, waiting for a round-trip delay (corresponding to the delay of ACK and SET-INFO messages) and some processing time t for the client to define the new CONTROL information. By the time the client receives $2f$+1 INFO messages (acknowledgements for SET-INFO) from the replicas, the new Speculative phase starts in a new view.

[5]There is some back-off scheme here such that all replicas can receive INIT request from the same client.